Towards an Automatic Path from Java™ Bytecodes to Hardware Through High-Level Synthesis

João M P Cardoso
Universidade do Algarve/INESC
Phone: +355 1 3100288

Horácio C Neto
IST/INESC
Phone: +355 1 3100366

INESC, Rua Alves Redol 9, 1000 Lisboa, Portugal
{Joao.Cardoso, hcn}@inesc.pt
FAX: +351 1 3145843

Abstract
This article describes a new approach to synthesise dedicated hardware from a system specification using the Java language. The new compiler named GALADRIEL starts from Java classfiles produced from the initial Java specification and processes the system information in order to exploit the concurrency implicit in each method, so that it can be efficiently implemented by multiple hardware and/or software components. The paper gives emphasis to the compiler techniques used to exploit the implicit concurrency and to the use of high-level synthesis to generate the hardware models from the Java bytecodes information.

1. Introduction
The Java language [1] has features that make it suitable to be adopted as an embedded systems specification language [2], as a generic programming language [3], and also as the Internet language. It has syntax mechanisms to support concurrency (critical to embedded systems), is architectural independent ("write once, run everywhere"), is a simple object-oriented language, is strongly typed, contains exception handling mechanisms, is safe and robust, is dynamic, etc.

However it has a few drawbacks, namely the low performance of the execution code. Some efforts to approximate the execution time of Java programs to similar C/C++ programs take advantage of: optimisation techniques integrated in JIT (Just-In-Time) compilers [4], optimisation of bytecodes with advanced compiler techniques [5][6], loop parallelization [7] when real multithreaded is supported, and development of direct execution support (Java microprocessors) [8]. Nevertheless all of these efforts can be substantially improved with the integration of HW/SW codesign and High-Level Synthesis (HLS) [9] techniques. Typical HW/SW codesign approaches start from a SW specification and migrate SW blocks to HW in order to satisfy timing constrains [10] or to accelerate a generic application in a host by means of HW boards [11]. Others begin with an HW specification and migrate to SW some of the HW blocks such that the timing requirements of the specification remain satisfied [12].

Our approach starts with an abstract model of computation and uses co-synthesis techniques supported by an HLS system to accelerate the Java applications. The front-end is the GALADRIEL compiler, which explores the parallelization of SW blocks, and the translation from bytecodes to VHDL.

This paper describes some of the work done so far and indicates future directions for the research involved. In Section 2 we summarise the Java technology. Then, Section 3 describes the computations and transformations performed by the compiler. The automatic path from the specification to the HW implementation is described in Section 4. Finally in Section 5, an example is presented and the results obtained are shown.

2. The Java Technology
Each Java class of a given application is compiled to a virtual machine designated by JVM (Java Virtual Machine) [13]. The compilation results of each Java class reside in a file named Java classfile which contains a constant pool (with symbolic references, constant strings, and information about the class) and the bytecodes for each method, among other information [13]. The JVM has about 200 instructions that can be distributed over 30 functionality type groups and each instruction has an 8-bits opcode. The JVM is stack-oriented, with an operand’s stack and local variables for each method. The operations get values from the operand stack and store the result in it. The operand stack is also used to pass arguments to methods and to receive the return result from a called method. There are instructions to transfer word contents from/to the local variables to/from the operand stack, instructions related to the manipulation and creation of objects, arrays, and the information encapsulated within it, control-flow instruc-

---

1 This work has been partially supported by the Ph.D. program of the Prodep 5.2 action and the program Praxis XXI under the scope of Project PRAXIS/2/2.1/TIT/1643/95.
sections (like if, goto, and switch types), jumps to subroutines handling exceptions, method invocation, etc. The JVM supports multithreaded applications by means of two bytecode instructions to synchronise blocks of code: monitorenter and monitorexit. Additional support is provided by the java.lang.Thread class [1]. Object access synchronisation across multiple threads is specified with the word synchronised and treated with the two previous bytecode instructions. The classfiles can be executed using an interpreter (like java from Sun™) or a JIT compiler (compiles on-the-fly bytecodes to native code); directly with a Java microprocessor; or with an off-line compilation to native code.

3. The GALADRIEL Compiler

From the set of classfiles of a given application the GALADRIEL extracts the needed information and represents it in an internal model that contains a DAG (Directed Acyclic Graph), where each node represents a method and each edge represents the method invocation hierarchic flow (see Fig. 1). This graph is called Method Calling Sequence Graph (MCSG) and its root is the main() method. Each run() method in the MCSG, which correspondent class extends the Thread class or implements the Runnable interface, is tagged as concurrent.

From the bytecodes of each method in the MCSG, the GALADRIEL extracts the instructions and performs the transformations shown in Fig. 2. For each method a control flow graph (CFG) [14], \( G(V, E) \), is generated that consists in a directed graph with each vertex \( V \) representing a JVM instruction and edges \( E \) representing the execution dependence between instructions. Each CFG includes a START and an END vertex (all other vertices are included in a path between the two vertices). There is an edge between two vertices \( (V_i, V_j) \) if and only if the instruction in \( V_j \) can be executed immediately after the instruction in \( V_i \). This model of CFG includes switch type vertices (like the tableswitch and lookupswitch of the JVM [13]). The compiler may optionally output to a file an ascii representation of each graph to be viewed by the DOTTY tool [15].

To analyse the data and control dependencies of a method at basic block level, the CFG after the computation of the leaders is transformed in the CFGBB, in which the new vertices are basic blocks (BBs) [14] of JVM instructions.

With this graph the dominators set (DS) is computed by the iterative algorithm described in [14]. From that the compiler identifies the back edges, the BBs forming natural loops, and groups them in a special vertex after the identification of the related loops information (bounded, unbounded, counter variable, incremental value, initial value, and final condition). In the presence of nested loops the set of basic blocks of an outer loop contains the equivalent set of the inner loop and so on. Like in [16] the CDG (Control Dependence Graph) is created from the CFGBB and the pos-dominators set (PDS). The PDS is constructed with the same algorithm of the DS in the reverse traversing order of the CFGBB. To construct the DDG (Data Dependencies Graph) the GALADRIEL first determines the reaching definitions [14], from the local variables accessed by the JVM instructions in order to compute the data-dependencies over local variables, and then computes the stack-dependencies (SD). From that it computes the Use-Definition (UD) chain for each use of a local variable that indicates whose local variable’s definitions reach an usage. From the UD it constructs the Definition-Use (DU) chain for each definition of a local variable, that indicates the set of usages that are reached by each definition of a variable.

A BB, is data-dependent of BB, if it uses a local variable defined by the last time in BB, The SD are determined traversing the instructions and storing for each one the state of the stack that consists in the information about the stack contents (operands, types, instruction that push the operand, etc.). This is possible because the stack model of the JVM is invariant [13], i. e. the stack contains always the same number and type of operands in the execution flow of each instruction of the CFG. After this, for each BB our algorithm only examines the stack-state of each instruction in the BB to determine its dependencies. If all the stack-states that enter in a BB exit it then this BB does not depends of another BB unless any instruction that uses stack operands without pop them (like the dup instruction) is involved. In this case the algorithm examines the operands involved. The data-dependencies caused by an utilisation of a variable or stack-operand that stores a constant do not generate any data-dependencies indeed. The DDG is the union of the two generated dependencies. From the DDG and the

---

**Fig. 1.** The Method Calling Sequence Graph and the methodology flow.

**Fig. 2.** Intra-procedural compiler phases.
CDG a CFG with possible concurrent BBs is constructed as described in [16]. The data-dependencies analysis of the definition and usage of object members and methods, and array elements are under development. An array image is associated with each created array and every element of the image will store information about the utilisation of the corresponding element of the original array. This information is precious to analyse the possibility of loop parallelisation (an important compiler technique since most times the loops included in a specification manipulate arrays). At the moment, the code segments that degrade the execution performance of the application need to be identified by the user with the help of a profiler tool. A library for inserting tags in the Java source code has been developed, which includes the Tag interface with methods to specify code segments (start, end, rate, etc) and two classes that implement it in order to specify timing constraints (maxtime and mintime). The compiler uses the calls to these methods to identify the user-defined code segments and the constraints specified. The use of this library permits the analysis of the performance of the code between tags in the host machine, by the execution of an interpreter or JIT. The GALADRIEL eliminates all the references to this library in the bytecode. This allows the execution of the application need to be identified and executed in a Sun SPARC 10 under SunOS 4.1.4 with the Java interpreter from JDK 1.1.5 [18]. The compiler method (doSobel()) has 48 lines of Java, compiled to 430 bytes of bytecode. It has 267 instructions grouped in 60 BBs (see Fig. 4). This algorithm performs two convolutions (with two nested loops each one) and has more 3 loops. The HW solutions, obtained with the use of the CADDY-II [19] HLS system, follows the DMA principle, stores the intermediate arrays in memory, and was mapped to the IMS Sea-Of-gates technology [20]. The back-end logical synthesis phase is done with the Synopsys Design Compiler [21].

### Table 1. Results of the doSobel() method.

<table>
<thead>
<tr>
<th>Solution</th>
<th>Code Segment</th>
<th>Input image (m x n pixels)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SW</td>
<td>doSobel()</td>
<td>16 x 16 64 x 64</td>
</tr>
<tr>
<td>1st Conv.</td>
<td></td>
<td>1597 cycles 51709 cycles</td>
</tr>
<tr>
<td>HW @ 25 MHz</td>
<td>1st Conv.</td>
<td>1597 cycles 51709 cycles</td>
</tr>
</tbody>
</table>

The GALADRIEL executes in 11.1s and predicts that the three last loops can be executed in parallel. Moreover if
we consider the loop inter-iterations parallelization all the iterations of the three loops can be executed in parallel.

![CFG with BBs computed from the doSobel() method](image1)

Fig. 4. The basic block’s doSobel() CFG and the obtained CFG from the identification of the parallelizable BBs (right inferior corner).

6. Conclusions

We have described the techniques used in order to provide an HW performance acceleration of Java bytecode. The developed GALADRIEL compiler allows an efficient front-end to HLS systems starting from a system Java specification. It has the capability to determine intra-procedural implicit parallelism, which is an effective way to exploit HW/SW trade-offs, specially when the overall system consists of multi-components, thus having the possibility to map the parallel blocks into independent devices. All the code of the compiler has been developed in Java, assuring therefore its portability, and could be embedded in a Web browser. The current version of the GALADRIEL consists approximately of 7000 lines of Java. This work is one of the first endeavours, known by us, that attempts to use codesign techniques to accelerate Java applications.

The object-oriented specification encourages the use of many methods. Hence an inter-procedural (global) analysis is under development. The migration of methods and/or objects entirely to HW is feasible because typically the fan-in of the methods is small and the object-data can be stored in local memory. Because of the dynamic nature of the language, a static analysis of the memory object allocation like in [2] is performed whenever possible.

The use of the Java classfile intermediate representation, makes possible to support multi-language specifications.

References